Volume 08 Issue 01 Published, February 18, 2004 ISSN 1535-864X

Intel® Technology Journal

Intel® Pentium® 4 Processor on 90nm Technology

Support for the Intel® Pentium® 4 Processor with Hyper-Threading Technology in Intel® 8.0 Compilers

A compiled version of all papers from this issue of the Intel Technology Journal can be found at:
http://developer.intel.com/technology/itj/index.htm

Support for the Intel® Pentium® 4 Processor with Hyper-Threading Technology in Intel® 8.0 Compilers

Kevin B. Smith, Enterprise Platforms Group, Intel Corporation
Aart J.C. Bik, Enterprise Platforms Group, Intel Corporation
Xinmin Tian, Enterprise Platforms Group, Intel Corporation

Index words: Compilers, Intel Pentium 4 processor, Optimization, OpenMP, Hyper-Threading Technology, Vectorization

ABSTRACT

Intel's 8.0 compilers enable software developers to take advantage of the new architectural and micro-architectural features of the latest Intel® Pentium® 4 processor with Hyper-Threading Technology. This paper describes the support for both automatic optimization techniques and programmer-controlled methods of achieving high performance using the Intel 8.0 C++ and FORTRAN compilers. Details of both the automatic and programmer-controlled optimization techniques are presented. Results show that use of this compiler can significantly speed up software running on this new processor.

INTRODUCTION

The latest Intel Pentium 4 processor with Hyper-Threading Technology contains new features, both architectural and micro-architectural, which Intel's 8.0 compiler family uses to significantly increase software performance. This paper shows how the Intel 8.0 C++ and FORTRAN compilers enable software developers to take advantage of the new features of this latest Intel processor.

The latest Intel Pentium 4 processor implements a set of new instructions called the Streaming-SIMD-Extensions 3 (SSE3). We present an overview of these new instructions. Similarly, we discuss the new micro-architectural features and changes that affect the compiler.

Intel's compilers support both automatic optimization techniques and programmer-controlled methods to achieve high-performance software. The compiler provides two automatic optimization techniques to gain performance from recompilation: vectorization and advanced instruction selection. We describe new vectorization capabilities focusing on how these improve performance on the new processor. Then we present a section on advanced instruction selection providing details on both the implementation of complex data operations using the new SSE3 instructions, and on how micro-architectural changes affect instruction selection for other operations. The mapping of complex operations onto SSE/SSE2/SSE3 instructions is also shown, and we contrast this with the implementation of complex data types when generating code for Intel processors that do not support the SSE3 instructions. Next, we discuss how changes to the compiler's advanced instruction selection are motivated by the processor's micro-architectural changes. Experimental results show that together these improvements to automatic optimization techniques can speed up software by up to 25%.

Intel's compilers also offer programmers the ability to leverage performance features of Intel's processors by making changes to their source code. There are two kinds of source-level changes that the user can perform to take advantage of this new processor's performance features: direct insertion of SSE3 instructions and insertion of OpenMP* [7, 8] directives. The direct insertion of SSE3 instructions can be done either with intrinsic functions, which map directly to SSE3 instructions, or with inline assembly code containing SSE3 instructions. Both methods are discussed in this paper. Insertion of OpenMP

® Intel and Pentium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other brands and names are the property of their respective owners.

directives allows the programmer to take advantage of improved Hyper-Threading (HT) Technology support in this latest Intel Pentium 4 processor. Results are presented showing that our OpenMP implementation works with the improved HT Technology to produce significant performance improvements.

We conclude by summarizing the ways in which Intel's 8.0 compilers enable programmers to extract maximum performance from this latest Intel processor, and we review the performance improvements that have been achieved using these methods.

Throughout the paper, assembly examples follow the conventional format (see [13]).

NEW FEATURES

The latest Intel Pentium 4 processor contains both architectural and micro-architectural changes that the compiler uses to increase software performance.

New Instructions

The Streaming-SIMD-Extensions 3 supports 11 new instructions that can be used by the compiler to boost software performance: *addsubpd*, *addsubps*, *haddpd*, *haddps*, *hsubpd*, *hsubps*, *movddup*, *movshdup*, *movsldup*, *fisttp*, and *lddqu*. The new addsubpd and addsubps instructions allow one vector element to be subtracted while its next vector element is added. The haddpd, haddps, hsubpd, and hsubps instructions provide the capability to add or subtract horizontally within a vector, which enables more efficient clean-up code at the end of vectorized reduction loops. The movddup, movshdup, and movsldup instructions allow duplication of certain data elements into a vector. The lddqu instruction is a more efficient form of movups, useful when a memory load is likely to cross a cache-line boundary. Finally, the fisttp instruction provides a more efficient implementation of floating-point to integer conversion.

Micro-Architectural Changes

In addition to providing new instructions, the latest Intel Pentium 4 processor also has micro-architectural changes that the compiler can use to boost software performance. The latency of the multiply, shift, rotate, and some SSE/SSE2 instructions has been decreased. This reduced latency makes it more desirable than it used to be for the compiler to use these instructions. The processor's ability to forward stored data to a subsequent load that overlaps the store has also been improved. This allows more aggressive vectorization, since there is less probability that inefficiencies due to store mis-forwarding occur as a side effect. There are also many micro-architectural changes, which significantly improve Hyper-Threading Technology performance, and increase the opportunities for software performance improvement as a result of threading of an application.

AUTOMATIC OPTIMIZATION TECHNIQUES

New compiler options enable the generation of SSE3 instructions and tuning specifically for the micro-architectural changes in the latest Intel Pentium 4 processor. On Microsoft Windows* platforms, the /QxP option [11] directs Intel's compilers to use SSE3 and to tune for the new processor (for Linux* platforms the –xP option [12] has the same effect). Additional options are available (/QaxP [11] for Windows and –axP [12] for Linux) that direct the compiler to generate special high-performance copies of functions that can be sped-up using automatic techniques targeted for the new processor. These high-performance function copies will only be executed on the new processor, while on older Intel processors a less optimized version will be executed. When these options are specified, two classes of automatic optimizations are used to improve software performance: vectorization and advanced instruction selection.

Vectorization

Multimedia extensions provide a convenient way to exploit fine-grained parallelism in an application. Because manually rewriting sequential software into a form that exploits multimedia extensions can be rather cumbersome, vectorizing compilers have proven to be necessary tools for making multimedia extensions easier to use. Details of the vectorization methodology used by the Intel® C++ and FORTRAN compilers are given elsewhere [2][3]. This section briefly discusses aspects of vectorization that are specific to exploiting SSE3 instructions.

Vectorization of Single-Precision Complex Data Types

A complex number $c \in \mathbb{C}$ has the form

$$\mathbf{c = x + y \cdot i}$$

where $x,y \in \mathbb{R}$ denote the real part and imaginary part, respectively. The C99 standard [5] and FORTRAN have built-in single-precision and double-precision complex data types that simplify programming with complex numbers by assigning the usual complex semantics to operators like + (addition), - (subtraction), * (multiplication), and / (division) when applied to operands with a complex data type. Some SSE3

* Other brands and names are the property of their respective owners.

instructions are particularly useful to vectorize these complex operations in an efficient manner. Consider, for example, the FORTRAN code shown below.

```
complex a(10), b(10), c(10)
do i = 1, 10
    a(i) = b(i) * c(i)
enddo
```

The data layout of the single-precision complex arrays interleaves the 4-byte real parts and 4-byte imaginary parts of all complex elements in the 80-byte chunk of memory allocated for each array. Since complex multiplication is defined as

$$c \cdot c' = (x \cdot x' - y \cdot y') + (x \cdot y' + y \cdot x') \cdot i$$

the Intel compiler can translate the DO-loop shown above into the following sequence of SSE and SSE3 instructions.

```
L:
movaps     xmm0,    XMMWORD PTR b[eax]
movsldup   xmm2,    XMMWORD PTR c[eax]
mulps      xmm2,    xmm0
movshdup   xmm1,    XMMWORD PTR c[eax]
shufps     xmm0,    xmm0, 177
mulps      xmm1,    xmm0
addsubps   xmm2,    xmm1
movaps     XMMWORD PTR a[eax], xmm2
add        eax,     16
cmp        eax,     80
jb         L
```

Since vector iterations process two complex data elements at one time, the loop only iterates five times. Furthermore, fine-grained computational overlap between operations on the real and imaginary parts of each complex data element is obtained.

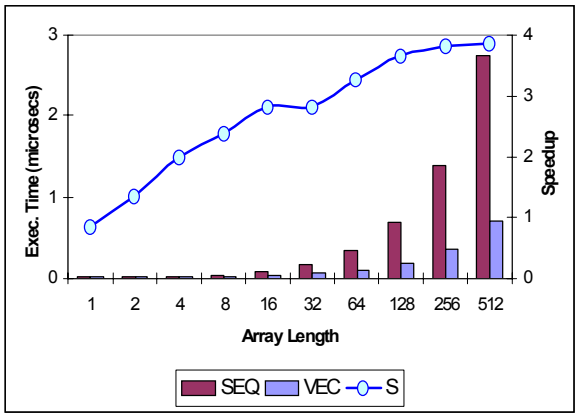


Figure 1: Single-precision complex vectorization speedup

Figure 1 plots the execution times (in microseconds) of the FORTRAN loop shown above using a sequential FPU-based implementation (SEQ) and a vectorized implementation using SSE3 instructions (VEC) for various array lengths using a 2.8GHz Intel Pentium 4 processor with HT Technology. Execution times were obtained by running the kernel repeatedly and dividing the total runtime accordingly. The corresponding speedup (S) is shown in the same figure using a secondary y-axis. Clearly, vectorizing single-precision complex operations by means of SSE3 instructions already pays off for arrays of length two or more, with the speedup going up to almost four.

Vectorization for SSE3 Idioms

Some SSE3 instructions enable a slightly more efficient implementation of constructs that were already vectorized by previous versions of Intel's compilers. The instruction haddps, for instance, simplifies accumulating partial sums after a vectorized reduction. The following C example

```
float a[100], red;
…
red = 0;
for (i = 0; i < 100; i++) {
   red += a[i];
}
```

is vectorized as shown below, where the loop body exploits four-way SIMD parallelism to implement the sum-reduction. The post loop clean-up code uses two horizontal add instructions to reduce the four partial sums back into one final sum again (formerly, this required six instructions).

```
    pxor      xmm0, xmm0
L:  addps     xmm0, XMMWORD PTR a[eax]
    add       eax,  16
    cmp       eax,  400
    jb        L
    haddps    xmm0, xmm0
    haddps    xmm0, xmm0
    movss     DWORD PTR red,  xmm0
```

Other SSE3 instructions provide a more compact way to implement some frequently occurring data rearranging instruction sequences.

Improved store-to-load forwarding in the latest Intel Pentium 4 processor also allows the Intel 8.0 compilers to become more aggressive in exploiting fine-grained SIMD parallelism in straight-line code. As an example, the structure shown below

```
struct {
  double x;
  double y;
} v;

...
v.x  += 1.0;
v.y  += 2.0;
```

can be mapped to only a few SSE2 instructions.

```
    movapd   xmm0, XMMWORD PTR v
    addpd    xmm0, const_1.0_2.0
    movapd   XMMWORD PTR v, xmm0
```

Such straight line code vectorization offers the compiler many more opportunities to improve the performance using SSE/SSE2/SSE3 instructions than more traditional loop-oriented vectorization. Collapsing fine-grained parallelism into enclosing loops exposes more iterations to a vector loop. This increases the probability that loops can be effectively vectorized and enables the compiler to apply more advanced methods, such as dynamic loop peeling for alignment or dynamic data dependence testing (see [3] for details). The two statements in the following loop-body, when taken in isolation, do not expose enough parallelism to enable use of 4-way parallel SSE instructions.

```
struct {
  float x;
  float y;
} a[100];

for (i = 0; i < 100; i++) {
   a[i].x = 0;
   a[i].y = 0;
}
```

However, when collapsed into the loop the full potential of SSE instructions can be exploited, as shown below.

```
    pxor    xmm0, xmm0
    xor     eax, eax
L:  movaps  XMMWORD PTR a[eax], xmm0
    add     eax, 16
    cmp     eax, 800
    jb      L
```

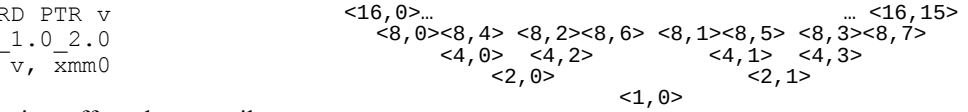
Interprocedural Alignment Analysis

Like most instructions, multimedia instructions operate more efficiently when memory operands are aligned at their natural boundary, i.e., 64-bit memory operands should be 8-byte aligned and 128-bit memory operands should be 16-byte aligned. In order to obtain aligned access patterns in vector loops, the Intel compilers perform advanced static and dynamic alignment analysis and an enforcement method, as described elsewhere [2][3]. The Intel 8.0 compilers further extend this support with interprocedural alignment analysis. This analysis consists of finding maximal values 2^n in a mapping ALIGN such that for a function f(), the value

$$\text{ALIGN}(f, p) = \langle 2^n, o \rangle \quad \text{with } 0 \le o < 2^n$$

denotes that all actual arguments that are associated with formal pointer/call-by-reference argument p evaluate to an address A that satisfies A mod 2^n = o.

By defining an alignment lattice of the form



with the meet operator going toward bottom element <1,0> (i.e., arbitrary alignment), and jump functions that use modulo arithmetic to correlate incoming formal arguments with any outgoing expression used as actual argument, the problem can be solved similar to the algorithm given in [4] for interprocedural constant propagation. More details can be found in [3]. A similar approach to propagating alignment information within a function was proposed by Larsen et. al. in [6].

More precise knowledge on the alignment of data structures can substantially increase the effectiveness of using SSE3 instructions. The compiler's ability to perform interprocedural alignment analysis is partially responsible for the large performance improvement that is seen in the results section for the 168.wupwise benchmark.

Advanced Instruction Selection

As stated earlier, both FORTRAN and C99 support basic data types used for representing complex numbers. Advanced instruction selection uses the new SSE3 instructions to implement these complex types' basic operations. Additionally, micro-architectural changes create opportunities for advanced instruction selection optimizations to tune for the latest Intel Pentium 4 processor. Both of these forms of instruction selection will be covered in the following sections.

Implementing Complex Operations with SSE3

As described previously, the representation of the complex data types interleaves the real and imaginary parts in memory. The real portion occupies memory at the lowest address and the imaginary portion occupies memory immediately above the real portion. With the addition of SSE3 instructions, operations on complex data map well onto the vector instruction sets provided on the latest Intel Pentium 4 processor.

The C99 Standard [5] defines many operations and library routines that perform computation on complex data items. In addition to the basic operations mentioned previously, other complex operations are provided by library routines. These additional operations include complex conjugate (conj), extract real part (creal), and extract imaginary part (cimag). In the Intel compilers' intermediate representation, both single- and double-precision complex data types and operations are directly represented and optimized when generating code targeting the latest Intel Pentium 4 processor. At the transition from machine-independent optimization to machine-specific code generation, the complex operations are translated directly

into SSE, SSE2, and SSE3 instructions. This occurs for the basic operations and for the simpler library routines. We discuss how these operations are translated in the next section.

Complex Loads and Stores

The most basic of operations on complex data are loading and storing. For single-precision complex data, the movlps instruction is used to move the data into the lower two single-precision vector elements of an xmm register. This same instruction is also used for storing single-precision complex data. For double-precision complex data, the movapd instruction will be used to load and store the data to/from an xmm register. This instruction can only be used if the effective address for the load/store is known to be 16-byte aligned. In the event that the compiler cannot prove alignment of the data, then a sequence of movlpd, movhpd instructions will be used to perform the load/store. The ability to use the aligned forms of these instructions has a significant impact on the performance of the resultant code, as discussed previously.

Creation of Complex from Real, Imaginary Parts

Complex numbers are often created from separate expressions for the real and imaginary parts. This is done using the unpcklps or unpcklpd instructions. For the double-precision complex creation operation, if both expressions are loaded directly from memory, then this can be more efficiently done with the movlpd, movhpd instructions. A common operation of creating a double-precision complex with an imaginary part of 0.0 from a real expression is done with an xorpd instruction, followed by a movsd instruction. When the real value is a memory expression, the xorpd is unnecessary as movsd from memory zeros the upper vector element.

Complex Addition and Subtraction

The operations of complex addition and subtraction are defined as adding or subtracting the real and imaginary portions of the data. These operations map straightforwardly onto the addps/subps and addpd/subpd instructions, respectively, for single- and double-precision complex data.

Complex Conjugate

The complex conjugate operation is defined as leaving the real portion of the complex number unchanged, while negating the imaginary portion. The fastest implementation of this is done using the xorps/xorpd instructions with a constant having only the sign bit of the imaginary part set. This toggles the sign bit of the imaginary part, while leaving the real portion unchanged. The implementation of complex conjugate for a double-precision complex data object that resides in the xmm0 register is shown below.

```
_floatpack.1: DWORD 0x80000000
              DWORD 0x00000000
              DWORD 0x00000000
              DWORD 0x00000000
…xorpd xmm0, XMMWORD PTR _floatpack.1
```

Complex Multiplication

The definition of complex multiplication was given above in discussing vectorization of single-precision complex data types. Complex multiplication of two operands A and B of double-precision complex type is handled quite similarly. The code sequence is shown below.

```
movapd   xmm0, XMMWORD PTR A
movddup  xmm2, QWORD PTR   B
mulpd    xmm2, xmm0
movddup  xmm1, QWORD PTR   B+8
shufpd   xmm0, xmm0, 1
mulpd    xmm1, xmm0
addsubpd xmm2, xmm1
movapd   XMMWORD PTR C, xmm2
```

As can be seen above, the movddup and addsubpd SSE3 instructions make the implementation of complex multiplication quite efficient. In contrast, the code generated for this same operation when targeting older Intel Pentium 4 processors is shown below.

```
movsd     xmm3, QWORD PTR A
movapd    xmm4, xmm3
movsd     xmm5, QWORD PTR A+8
movapd    xmm0, xmm5
movsd     xmm1, QWORD PTR B
mulsd     xmm4, xmm1
mulsd     xmm5, xmm1
movsd     xmm2, QWORD PTR B+8
mulsd     xmm0, xmm2
mulsd     xmm3, xmm2
subsd     xmm4, xmm0
movsd     QWORD PTR C, xmm4
addsd     xmm5, xmm3
movsd     QWORD PTR C, xmm5
```

As can be seen this sequence uses the scalar SSE2 instructions and isn't as efficient in either code size or execution time.

Complex Real and Imaginary Part Extraction

Both the creal and cimag operations are generated using the vector instruction sets. The creal operation requires no instructions at all. The compiler simply uses scalar SSE/SSE2 instructions to operate on the low vector element in future computations. For the cimag operation, if the complex value is in memory, the compiler simply adjusts the memory address and size to refer to the imaginary part of the value. When the value is in an xmm register, then a movshdup or unpckhpd instruction is used to copy the imaginary value into the lowest vector element of the register. Thereafter, scalar SSE/SSE2

operations will be performed using the resulting value in the lowest vector element.

Partial Constant Propagation and Folding

The expansion of complex operations into vector instructions can result in some partially redundant operations. For example in the code below, a double-precision complex value A is multiplied by a scalar B that has been cast to a double-precision complex value. After the cast the complex representation of B has an imaginary part of 0.0.

```
C = A * (double _Complex)B;
```

The expansion of this multiplication would be as shown below.

```
movapd    xmm0, XMMWORD PTR A
movsd     xmm2, QWORD PTR   B
movapd    xmm1, xmm2
movddup   xmm2, xmm2
mulpd     xmm2, xmm0
unpckhpd  xmm1, xmm1
shufpd    xmm0, xmm0, 1
mulpd     xmm1, xmm0
addsubpd  xmm2, xmm1
movapd    XMMWORD PTR C, xmm2
```

The unpckhpd in the above sequence serves to duplicate the imaginary part into both high and low vector elements. However, the upper vector element of xmm1 can be proven to be zero, so the unpckhpd instruction just creates a vector of 0.0. This zero vector will be used as an operand to a mulpd; therefore, the mulpd will also produce a zero vector. The result of the mulpd is an operand of addsubpd. A source operand of a zero vector is an identity operand for addsubpd, passing through the destination operand unchanged. So after partial constant propagation and arithmetic simplification, the above code has been optimized into the code below.

```
movapd   xmm0, XMMWORD PTR A
movddup  xmm2, QWORD PTR   B
mulpd    xmm2, xmm0
movapd   XMMWORD PTR C, xmm2
```

Partial constant folding and arithmetic simplification allows extremely efficient generation of complex operations involving scalar values converted to complex, a frequent occurrence. In a micro-benchmark containing only complex multiplication by a scalar, this optimization improves execution time by 55%.

Maximal Use of SSE/SSE2/SSE3 Instructions

The latest Intel Pentium 4 processor has improved latency for some of the frequently used operations in SSE and SSE2 instructions. For example, cvtps2pd latency is improved from seven cycles in earlier Intel Pentium 4 processor implementations to three cycles. In addition, micro-architectural improvements have raised the overall performance of the SSE/SSE2 instruction sets. The Intel 8.0 compilers now use SSE/SSE2/SSE3 instructions for all possible floating-point operations when generating code targeted for the latest Intel Pentium 4 processor. This improves performance of many floating-point-intensive applications, particularly those where performance was limited by denormal exception processing. The improvement in denormal processing is a result of using the flush-to-zero (FTZ) and denormals-are-zero (DAZ) modes that are available only with the SSE, SSE2, and SSE3 instruction sets.

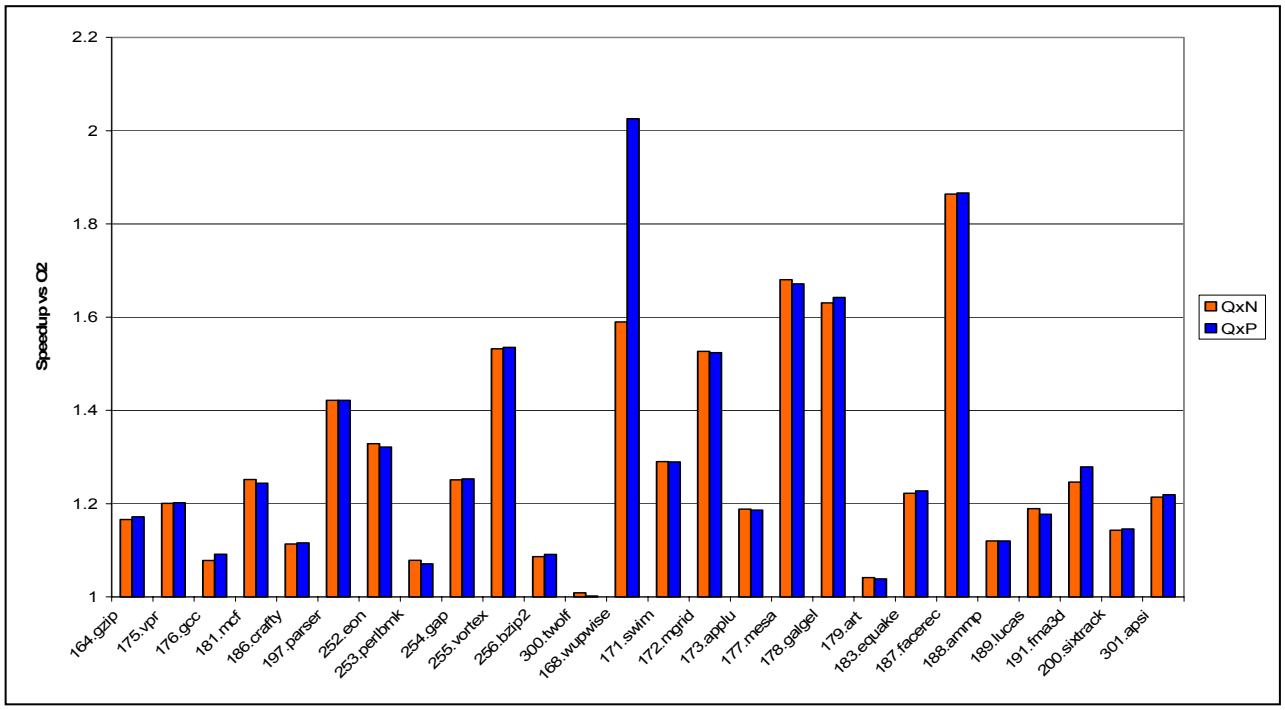


Figure 2: Processor-specific speedup of SPEC* CPU 2000 estimates (based on measurements on Intel internal platforms)

Lower Multiply/Shift Latency

The latency of the imul and shl instructions has been improved in the new Intel Pentium 4 processor. This caused changes to the compiler's heuristics for expansion of multiplication by compile-time constants. These changes result in more compact code at the same performance level. For example, when generating code for older Intel Pentium 4 processors, a multiply of edx by 68 produces

```
lea    ecx, DWORD PTR [edx+edx]
add    ecx, ecx
lea    eax, DWORD PTR [ecx+ecx]
add    eax, eax
add    eax, eax
add    eax, ecx
```

However, using compiler option /QxP, the code below is produced for the same operation.

```
mov        ecx, edx
shl        ecx, 6
lea        eax, DWORD PTR [ecx+edx*4]
```

The resulting code is smaller and has slightly better performance than the earlier sequence.

Fisttp Instruction Usage

The SSE3 fisttp instruction is useful for converting floating-point data to integer. For conversion from floating-point format to 32-bit integer or smaller data types the compiler will use the cvtsd2si or cvtpd2dq instructions. However, for converting from floating-point format to 64-bit signed or unsigned integer format, the fisttp instruction is most efficient, and the compiler will use the instruction in that circumstance. This is illustrated by the following snippet of C code.

```
__int64 llfunc(double in)
{
    return (__int64)(in);
}
```

The code generated for this by the Intel 8.0 C++ Compiler using the –QxP –O2 options follows.

```
_llfunc PROC NEAR
        sub       esp, 20
        fld       QWORD PTR [esp+24]
        fisttp    QWORD PTR [esp]
        mov       eax, DWORD PTR [esp]
        mov       edx, DWORD PTR [esp+4]
        add       esp, 20
        ret
```

* Other brands and names are the property of their respective owners.

For comparison, the code below is generated for the same code when using the –QxN –O2 options. Note the use of the fnstcw and fldcw instructions that are necessary to assure that rounding is done towards 0 as required by the C language.

```
_llfunc PROC NEAR
        sub       esp, 20
        fld       QWORD PTR [esp+24]
        fnstcw    [esp+8]
        movzx     eax, WORD PTR [esp+8]
        or        eax, 3072
        mov       DWORD PTR [esp+16], eax
        fldcw     [esp+16]
        fistp     QWORD PTR [esp]
        fldcw     [esp+8]
        mov       eax, DWORD PTR [esp]
        mov       edx, DWORD PTR [esp+4]
        add       esp, 20
        ret
```

The code using the new fisttp instruction is 3 times faster than the code using the older code sequence as measured on a micro-benchmark doing only conversion of double-precision floating-point values to signed 64-bit integers.

SPEC* CPU2000 Performance Results

To give a flavor of the impact of processor-specific optimizations, some performance results are given for SPEC* CPU2000 [9]. This industry-standardized benchmark suite consists of 14 floating-point and 12 integer C/C++ and FORTRAN benchmarks that are derived from real-world applications. The graph in Figure 2 shows the speed-ups obtained on a 2.8GHz Intel Pentium 4 processor with HT Technology for each of the SPEC CPU2000 benchmarks. The bars denoted QxP represent the ratio of the performance of executables obtained using high-level, interprocedural, profile-guided, and processor-specific optimizations for the latest Intel Pentium 4 processor with HT Technology processor (`-O3 -Qipo -Qprof_use -QxP`) compared with the performance of executables that result when using default optimizations (`-O2`). Similarly, the bars denoted QxN represent the ratio of the performance of executables obtained using high-level, interprocedural, profile-guided, and processor-specific optimizations for older Intel Pentium 4 processors (`-O3 -Qipo -Qprof_use -QxN`) compared with the performance of the default executables. The results reveal the advantages of the latest processor-specific optimizations particularly for programs such as 168.wupwise, where performance is highly dependent on the speed of complex arithmetic. For 168.wupwise, this results in a 25% improvement compared to older Intel Pentium 4 processor-specific optimizations, and a 2X performance improvement compared to default (`-O2`) level of optimization. The results also show that significant performance improvements can be obtained using the latest Intel Pentium 4 processor even when using processor-specific optimizations (`-QxN`) that are targeted at older Intel Pentium 4 processors.

PROGRAMMER-GUIDED OPTIMIZATION TECHNIQUES

In addition to automatic optimization techniques, the Intel 8.0 compilers also support programmer-controlled methods of improving software performance for the latest Intel Pentium 4 processor. The Intel C/C++ compilers support intrinsic functions, which the compiler maps directly to the Streaming-SIMD-Extensions 3 (SSE3). These intrinsics can be fully optimized by the compiler. The C/C++ compilers also support inline assembly code, and the SSE3 instructions are fully supported in inline assembly. Both the C/C++ and FORTRAN compilers support OpenMP. The programmer can insert OpenMP directives into the source programs to allow their applications to be threaded, thus taking advantage of the improvements to Intel Pentium 4 processor Hyper-Threading (HT) Technology [10].

SSE3 Intrinsics

The C/C++ compiler provides a set of intrinsic functions that the compiler maps directly into SSE3. Intrinsics allow the programmer to write low-level code using SSE, SSE2, and SSE3 without having to worry about issues such as register allocation or instruction scheduling, for which compilers are well suited. This allows programmers to concentrate on mapping their algorithms efficiently to these instructions, and it allows the compiler to fully optimize these instructions. Table 1 lists the intrinsics supported for the SSE3 and the instruction that the intrinsic maps to.

Table 1: SSE3 intrinsic to instruction mapping

Intrinsic Name	Instruction Generated
_mm_addsub_ps	addsubps
_mm_hadd_ps	haddps
_mm_hsub_ps	hsubps
_mm_moveldup_ps	movsldup
_mm_movehdup_ps	movshdup
_mm_addsub_pd	addsubpd
_mm_hadd_pd	haddpd
_mm_hsub_pd	hsubpd
_mm_loaddup_pd	movddup xmm, m64
_mm_movedup_pd	movddup reg, reg
_mm_lddqu_si128	lddqu

Inline Assembly Using SSE3

The Intel C/C++ compiler also supports inline assembly code in C/C++ source. All of the SSE3 instructions are supported in inline assembly. Examples of the inline assembly supported for each of the new instructions are shown below.

```
__asm {
    addsubpd xmm0, xmm1
    addsubpd xmm3, XMMWORD PTR mem
    addsubps xmm5, xmm1
    addsubps xmm4, XMMWORD PTR mem
    haddpd   xmm2, xmm7
    haddpd   xmm3, XMMWORD PTR mem
    haddps   xmm5, xmm6
    haddps   xmm0, XMMWORD PTR mem
    hsubpd   xmm2, xmm7
    hsubpd   xmm3, XMMWORD PTR mem
    hsubps   xmm5, xmm6
    hsubps   xmm0, XMMWORD PTR mem
    lddqu    xmm2, XMMWORD PTR mem
    movddup  xmm0, xmm1
    movddup  xmm2, QWORD PTR   mem
    movshdup xmm1, xmm0
    movshdup xmm3, XMMWORD PTR mem
    movsldup xmm1, xmm0
    movsldup xmm3, XMMWORD PTR mem
}
```

OpenMP-Based Multi-Threading

Due to the simplicity of OpenMP model, it has become the dominant high-level programming model to exploit the Thread-Level Parallelism (TLP) in various applications for shared-memory multi-threaded architectures. The Intel 8.0 C++/Fortran95 compilers support OpenMP [7,8] directive-guided parallelization [9], which significantly increases the domain of applications amenable to thread-level parallelism. An application example, shown below, uses OpenMP parallel sections to exploit the TLP of Audio-Visual Speech Recognition (AVSR) through functional decomposition.

```
…
#pragma omp parallel sections default(shared)
{
  #pragma omp section
    { DispatchThreadFunc( &AVSRThData; } // data input and dispatch
  #pragma omp section
    { AudioThreadFunc( &AudioThData ); }        // process audio data
  #pragma omp section
    {  VideoThreadFunc( &VideoThData );   }      // process video data
  #pragma omp section
      {   AVSRThreadFunc( &AVSRThData );  } // perform avsr
}
…
```

From the AVSR code sample shown above, we can see the *omp section*-1 invokes the call for data input and dispatching, the *omp section*-2 invokes the call to process the audio data, the *omp section*-3 invokes a call to process video data, and the *omp section*-4 invokes a call to do AVSR, so the performance gain is obtained by mapping four *sections* onto different logical processors to fully utilize processor resources based on HT Technology. In the next two sub-sections, we present optimizations used by Intel 8.0 compilers to tune performance for HT Technology.

Preloading with Aggressive Code Motion

When two threads are sharing data, it is important to avoid false sharing. An example of false sharing occurs when a private and a shared variable are located within the cache line size boundary (64 bytes) or sector boundary (128 bytes). When one thread writes the shared variable, the "dirty" cache line must be written back to memory and updated for each processor sharing the bus. Subsequently, the data are fetched into each processor 128 bytes at a time, causing previously cached data to be evicted from the cache on each processor. False sharing incurs a performance penalty when two threads run on different physical processors, due to cache evictions required to maintain cache coherency, or on logical processors in the same physical processor package, due to memory order machine clear conditions. In this section, we present an optimization-aggressive code motion that preloads all read-only shared memory references into register temps from inside of a region/loop/section to outside of a region/ loop/section, if a memory reference is proven to be a read-only memory reference based on the compiler's load-store analysis and memory disambiguation. The preloading code is moved right after the T-entry. For example, in Figure 3, the memory references of the dope-vector base-address, array lower bound, and stride are lifted to the outside of the loop and pre-loaded into a register temp *t0*, *t1*, and *t2*. Additionally, the memory references of *dv_ptr*-

>baseaddr, *dv_ptr->lower*, and dv*_ptr->stride* are replaced by register temp *t0*, *t1*, and *t2*, respectively.

The benefit of this optimization is that it reduces the data false sharing and avoids the performance penalty. This reduces the overhead of memory de-referencing, since the value is preloaded into a temporary register for the frequent read operations. The second benefit is that it enables advanced optimizations such as vectorization, if the memory de-references in array subscript expressions are lifted outside the loop. For example, in Figure 3, the address computation of array involves the memory de-references of the member *lower* and *extent* of the dope-vector, the compiler lifts the memory de-references of *lower* and *extent* outside the *m*-loop, because the compiler is able to prove that all references to members of the array's dope-vector are *read-only* within the parallel *do* loop. In general, this aggressive code motion enables a number of high-level optimizations such as loop unroll-and-jam, loop tiling, and loop distribution as well. This has resulted in good performance improvements in real applications.

```
real allocatable:: w(:,:)
… …
!$omp parallel do shared(x), private(m,n)
do  m=1, 1600                !! Front-End creates a dope-vector for allocatable
   do n=1, 1600              !! array w
      w(m, n) = …      ➔     dv_baseaddr[m][ n] = …
   end do
end do
… …
T-entry(dv_ptr …)   !! Threaded region after multithreaded code generation
    … …
    t0 = (P32 *)dv_ptr->baseaddr    // dv_ptr is a pointer that points
    t1 = (P32 *)dv_ptr->lower       //  to the dope-vector of array w
    t2 = (P32 *)dv_ptr->extent
    … …
    do prv_m=lower, upper
        do prv_n =1, 1600           //   EXPR_lower(w(prv_m, prv_n)) = t1
          t3[prv_m][prv_ n] = …     //   EXPR_stride(w(prv_m, prv_n)) = t2
        end do
    end do
T-return
… …
```

Figure 3: An example of code motion

Auto-Dispatching Fast Lock Routines

Efficient reduction of bus traffic and resource contention can significantly impact the overall performance of threaded applications on systems using the Intel Pentium 4 processor with HT Technology. One of the optimizations implemented in the 8.0 compilers reduces memory and bus contention by dispatching to fast lock routines in the Intel OpenMP runtime library based upon an auto-cpu dispatching mechanism. This allows generated code to query the thread *id* only once for each thread and re-use the thread *id* for invoking fast lock routines instead of the generic version of OpenMP lock routines in the source. The Intel compiler generates code that dynamically determines which processor the code is running on and chooses which version of the function will be executed accordingly. This runtime determination allows the library to take advantage of architecture-specific tuning without sacrificing flexibility by allowing execution of the same binary on older Intel IA32 processors that do not support some of the newer instructions. See the sample code in Figure 4, the *omp_set_lock* and *omp_unset_lock* routines are called 200,000 times inside a parallel loop.

```
… …
#pragma omp parallel for shared(x)
for (i=1,  i<200000; i++) {
   omp_set_lock(&lock0) ;
       x = x * foo(&x, i)  +  …
   omp_unset_lock(&lock0) ;
}

… …
tid = __kmpc_get_global_thread_num()       ; get thread id outside the loop
… …
$B1$25:
      mov      eax, DWORD PTR ___intel_cpu_indicator
      cmp      eax, 1                          ; CPU dispatch
      je       $B1$27
$B1$26:
      mov      eax, DWORD PTR [ebp+16]
      push     DWORD PTR [eax]              ; passing lock0
      push     ebx                          ; passing thread id
      push     OFFSET FLAT: _kmpc_loc     ; location info
      call     ___kmpc_set_lock             ; dispatching fast lock routine
$B1$49:
      add      esp, 12
      jmp      $B1$28
      ALIGN    4
$B1$27:
      mov      eax, DWORD PTR [ebp+16]
      push     DWORD PTR [eax]              ; passing lock0
      call     _omp_set_lock                ; call  generatic lock routine
$B1$50:
      pop      ecx
$B1$28:
      ... ...
```

Figure 4: Example of dispatching fast lock routines

Originally, the query of thread *id* was done inside the lock routine. Each lock routine accesses a shared data structure that is maintained by the runtime library, which can cause heavy bus traffic and resource contention due to frequent access to the shared data structure. With the fast version of lock routines, we get the thread *id* outside the loop and re-use it for invoking the fast-version of lock routines (e.g., *__kmpc_set_lock/__kmpc_unset_lock*) at runtime based on Auto-CPU-dispatch, which reduces the memory access contention and bus traffic significantly for all lock routines, while minimizing the overhead of those frequent queries of thread *id*. This is important to get performance gain on Intel hyper-threaded processors. In summary, the Intel 8.0 compiler generates code to call two versions of the lock functions. A generic version of the lock function is invoked that will run on any x86 processor. Another fast version would be tuned for the Intel Pentium 4 processor family enabled with HT Technology.

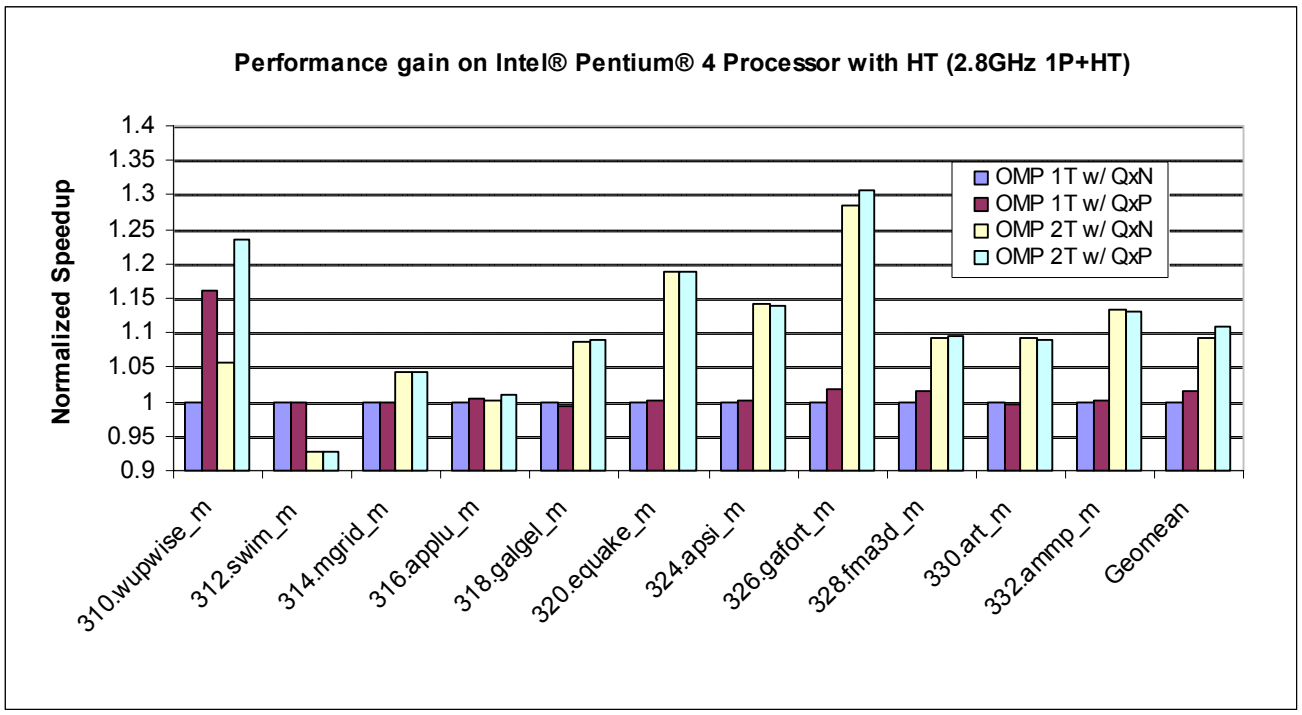


Figure 5: Speedup of SPEC* OMPM2001 estimates (based on measurements on Intel internal platforms)

SPEC* OMPM2001 Performance Results

The performance study of SPEC OMPM2001 benchmarks is conducted on a pre-production single physical processor system built with the latest Intel Pentium 4 processor with HT Technology, running at 2.8GHz, with 2GB memory, an 8K L1-Cache, and a 512K L2-Cache. The SPEC OMPM2001 is a benchmark suite that consists of 11 scientific applications.

Those SPEC OMPM2001 benchmarks target small- and medium-scale SMP multiprocessor systems, and the memory footprint reaches 2GB for several very large application programs. For our performance measurement, all SPEC OMPM2001 benchmarks are compiled by the Intel 8.0 C++/Fortran compilers with two sets of base options: -Qopenmp –Qipo –O3 –QxN (OpenMP w/ QxN) and –Qopenmp –Qipo –O3 –QxP (OpenMP w/ QxP). The normalized performance speed-up of the SPEC OMPM2001 benchmarks is shown in Figure 5. This demonstrates the performance gain attributed to Intel Hyper-Threading Technology and Intel 8.0 C++/Fortran compiler support by exploiting thread-level parallelism.

The performance scaling is derived from the baseline performance of single thread binary with OMP 1T w/ QxN, single thread execution under OMP 1T w/ QxP, and two threads execution under OMP 2T w/ QxN and OMP 2T w/ QxP, respectively.

As we can see, the multi-threaded code generated by the Intel compiler on a Intel Pentium 4 processor 1-CPU system enabled with Hyper-Threading Technology achieved a performance improvement of 4.2% to 28.6% (OMP 2T w/ QxN) on 9 out of 11 benchmarks except 316.applu_m (0.0%) and 312.swim_m (-7.3%). The 312.swim_m slowdown under the two-thread execution mode was well known: it is due to the fact that the 312.swim_m is a memory bandwidth bounded application. Overall, the geometric mean improvement with OMP 2T w/ QxN is 9.2% by running the second thread on the second logical processor. Furthermore, the geometric mean improvement with OMP 1T w/ QxP is 1.6%, due to the optimizations presented earlier. Under OMP 2T w/ QxP, we achieved a performance gain range from 4.2% to 30.6% for 9 out 11 benchmarks except the benchmarks 312.swim_m (-7.4%) and 316.applu_m (0.0%), and the geometric mean improvement was 10.9%. These performance results demonstrated that the multi-threaded codes generated and optimized by the Intel 8.0 compilers are very efficient when used with the support of the well-tuned Intel OpenMP runtime library.

CONCLUSION

Due to the complexity of modern microprocessors, compiler support has become an important part of obtaining high performance. The Intel 8.0 compilers provide full support for the rich features of the latest Intel Pentium 4 processor with Hyper-Threading Technology. The compilers perform automatic optimization of programs using vectorization and advanced instruction selection techniques that together can yield up to a 2x performance improvement compared to the default level of optimization. Additionally, the Intel compilers provide support for programmer-guided performance improvements by supporting the Steaming-SIMD-Extensions 3 (SSE3) directly using compiler intrinsics and inline-assembly, and by improving OpenMP directive support, enabling performance improvements of up to 30% for software using OpenMP directives. These capabilities allow software developers to use the Intel compilers to optimize the performance of their software for the latest Intel Pentium 4 processor.

More information on the Intel 8.0 compilers can be found at http://www.intel.com/software/products/compilers.

ACKNOWLEDGMENTS

The authors thank Zia Ansari, Mitch Bodart, Dave Kreitzer, Hideki Saito, and Dale Schouten for their contributions to the Intel 8.0 compilers and the KSL team for tuning the Intel OpenMP runtime library. The authors also thank Milind Girkar for his capable leadership of the IA32 compiler team.

REFERENCES

[1] Vishal Aslot et. al., "SPEComp: "A New Benchmark Suite for Measuring Parallel Computer Performance," in *Proceedings of WOMPAT 2001, Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science*, 2104, pp. 1-10, July 2001.

[2] Aart J.C. Bik, Milind Girkar, Paul M Grey, and Xinmin Tian, "Automatic Intra-Register Vectorization for the Intel Architecture," *International Journal on Parallel Processing*, 2001.

[3] Aart J.C. Bik, *The Software Vectorization Handbook: Applying Intel® Multimedia Extensions for Maximum Performance*, Intel Press, April 2004, http://www.intel.com/intelpress/sum_vmmx.htm.

[4] D. Callahan, K.D. Cooper, K. Kennedy, and L.M. Torczon, "Interprocedural Constant Propagation," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, New York, 1986.

[5] International Organization for Standardization, ISO/IEC 9899:1999. The standard can be obtained at http://www.iso.org/.

[6] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe, "Increasing and detecting memory address congruence," in *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[7] OpenMP Architecture Review Board, "OpenMP C and C++ Application Program Interface," Version 1.0, October 1998, http://www.openmp.org.

[8] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," Version 2.0, November 2000, http://www.openmp.org.

[9] Standard Performance Evaluation Corporation. SPEC CPU2000, http://www.spec.org/.

[10] Xinmin Tian, Aart J.C. Bik, Milind Girkar, Paul M. Grey, Hideki Saito, and Ernesto Su, "Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance," *Intel Technology Journal*, Vol. 6, Q1 2002.

[11] Intel® C++ Compiler for Windows Systems User Guide, http://www.intel.com/software/products/compilers/cwin/docs/ccug.htm

[12] Intel® C++ Compiler for Linux Systems User Guide, http://www.intel.com/software/products/compilers/clin/docs/ug/index.htm

[13] Intel Corporation. IA32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 2003. Manual available at http://developer.intel.com/.

AUTHORS' BIOGRAPHIES

Kevin B. Smith received his B.Sc. degree in Computer Science from Iowa State University in 1981. From 1981 to 1990 he worked at Tektronix, Inc on C and Pascal compilers and debuggers targeting 8086, Z8000, 68000, 68030, PDP-11, and VAX microprocessors. He joined Intel Corporation in 1990 working on compilers for the Intel i960® microprocessor, and in 1996 began working on compilers for the Intel Pentium® II, Pentium® III, and Pentium 4 processors. He is currently working in the Intel Compiler Lab as team leader for the IA32 code generator. His e-mail is kevin.b.smith at intel.com

Aart J.C. Bik received his M.Sc. degree in Computer Science from Utrecht University, The Netherlands, in 1992, and his Ph.D. degree from Leiden University, The Netherlands, in 1996. In 1997, he was a post-doctoral researcher at Indiana University, Bloomington, Indiana, where he conducted research in high-performance compilers for Java*. In 1998, he joined Intel Corporation where he is currently working on automatically exploiting multimedia extensions in the parallelization and vectorization group. His e-mail is aart.bik at intel.com

Xinmin Tian is currently working in the parallelization and vectorization group at Intel Corporation, where he works on compiler code generation and optimization for exploiting thread-level parallelism. He leads the OpenMP parallelization team. He holds B.Sc., M.Sc., and Ph.D. degrees in Computer Science from Tsinghua University. He was a postdoctoral researcher in the School of Computer Science at McGill University, Montreal. Before joining Intel Corporation he worked on a parallelizing compiler, code generation, and performance optimization at IBM. His e-mail is xinmin.tian at intel.com

*Other brands and names are the property of their respective owners.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm